CHAPTER 2

Construction of 2D Delaunay Triangulations

2.1 The Delaunay Kernel

Let DT_n be the Delaunay triangulation of a point set $S_n = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2$ that are in general position. We describe an incremental process allowing the insertion of a given point $p_{n+1} \in \Omega(S_n)$ into DT_n and to build the Delaunay triangulation DT_{n+1} of $S_{n+1} = \{p_1, \ldots, p_n, p_{n+1}\}$.

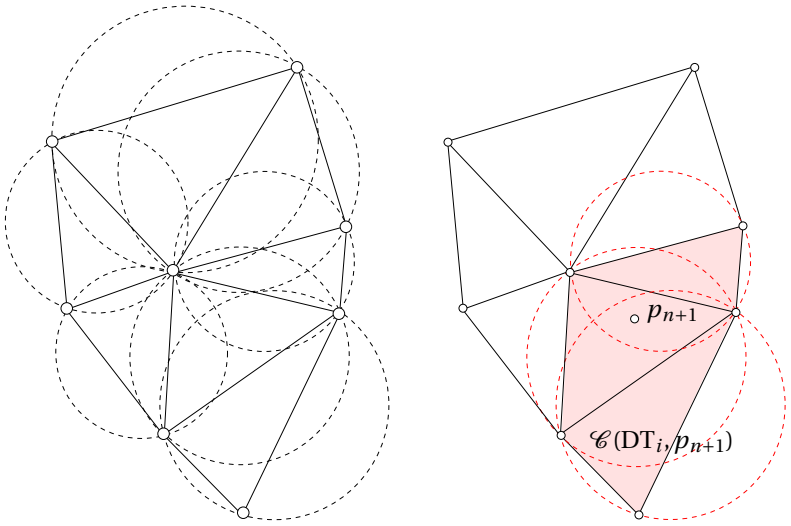

Figure 2.1: Delaunay triangulation T_n (left) and the Delaunay cavity $\mathscr{C}_p(\mathrm{DT}_n, p_{n+1})$ (right).

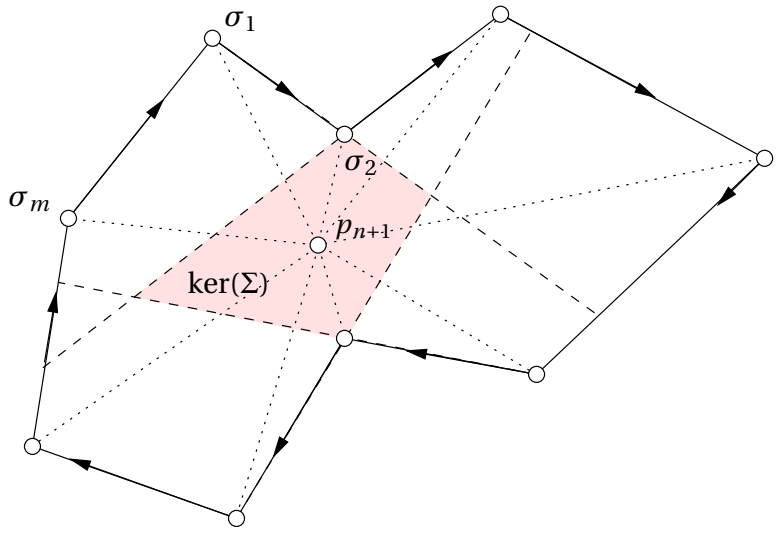

Figure 2.2: A star shaped polygon Σ and its kernel ker(Σ). All the corners σ_j, $1 \leq j \leq m$ of Σ are visible from any $x \in \ker(\Sigma)$.

Definition: The *Delaunay kernel* is the following procedure

$$\mathrm{DT}_{n+1} = \mathrm{DT}_n - \mathscr{C}(\mathrm{DT}_n, p_{n+1}) + \mathscr{B}(\mathrm{DT}_n, p_{n+1}). \tag{2.1}$$

The Delaunay cavity $\mathscr{C}(\mathrm{DT}_n, p_{n+1})$ is the set of all triangles whose circumcircles contain the new point p_{n+1} (see Figure 2.1) in consequense of what they are cannot belong to DT_{n+1}. The Delaunay ball $\mathscr{B}(\mathrm{DT}_n, p_{n+1})$ is a set of triangles that fill the polygonal hole that has been left empty while removing the Delaunay cavity $\mathscr{C}(\mathrm{DT}_n, p_{n+1})$ from DT_n.

In what follows, we will show that the Delaunay cavity $\mathscr{C}(\mathrm{DT}_n, p_{n+1})$ is star-shaped and that p_{n+1} belongs to its kernel. Then, we will explain how to build $\mathscr{B}(\mathrm{DT}_n, p_{n+1})$ in such a way that DT_{n+1} is a Delaunay triangulation.

2.1.1 Star shapeness

Consider a polygon Σ with m corners $\sigma_1, \ldots, \sigma_m$ that is bounded by m edges $\sigma_i, \sigma_{(i+1)\%m}$, $1 \leq i \leq m$.

Definition: The kernel ker(Σ) is the set of point $x \in \mathbb{R}^2$ that are visible to every σ_j i.e. the line segment $x\sigma_j$ them do not intersect any edges of the polygon.

The kernel ker(Σ) can be computed by intersection of the halfplanes that correspond to all oriented edges of the polygon (see Figure 2.2).

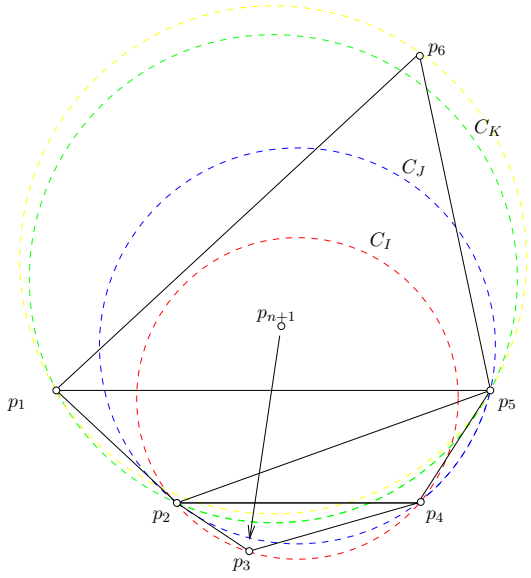

Figure 2.3: The delaunay cavity is star shaped.

2.1.2 The Delaunay Cavity

Definition: The Delaunay cavity $\mathscr{C}(T_n, p_{n+1})$ is the set of m triangles $\Delta_1, \ldots, \Delta_m \in \mathrm{DT}_n$ for which their circumcircle contains p_{n+1} (see Figure 2.1).

The Delaunay cavity contains the set of triangles that cannot belong to T_{n+1}. The region covered by those invalid triangles should be emptied and re-triangulated in a Delaunay fashion. The Delaunay cavity has some interresting properties.

Proposition 2.1.1 *The Delaunay cavity $\mathscr{C}(T_n, p_{n+1})$ is a non empty connected set of triangles which the union form a star shaped polygon with p_{n+1} in its kernel.*

Proof The proof is very similar to the one of proposition 1.3.4. Consider point p_{n+1} of Figure 2.3. Assume that p_{n+1} belongs to the circumcircle C_I of triangle $p_2p_3p_4$. Let's draw a line between $p_{n+1}l$ and p_3 which is the triangle that os the furthest away from p_{n+1}. If p_3 is our point of view, p_{n+1} is on the other side of p_2p_4. Point p_5 is outside C_I because triangle $p_2p_4p_5$ is a Delaunay triangle. Then The part of C_J which is on the orther side of p_2p_4 contains the part of C_J which is on the same side. This implies that triangle $p_2p_4p_5$ is invalid and is itself on the Delaunay cavity. We can continue that kind of argument starting with p_4, then p_2. Finally, triangle $p_1p_5p_6$ contains p_{n+1} and is obviously on the Delaunay cavity. So, any vertex of the boundary of the cavity can be seen by p_{n+1}, which proves the proposition. ∎

Property 2.1.1 *The Delaunay cavity $\mathscr{C}(T_n, p_{n+1})$ does not contain any point of S_n.*

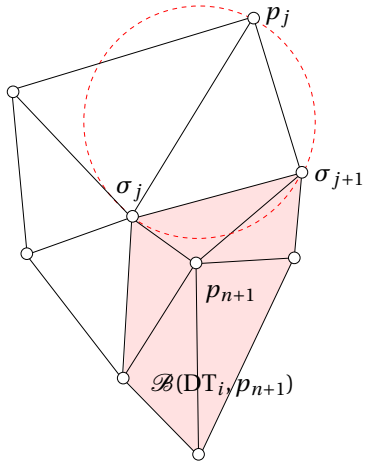

Figure 2.4: The Delaunay Ball.

Proof To do.

2.1.3 The Delaunay Ball $\mathscr{B}(\mathbf{DT}_p, p_{n+1})$

The Delaunay cavity $\mathscr{C}(\mathrm{DT}_n, p_{n+1})$ is star shaped and p_{n+1} belongs to its kernel. So, one possible solution for the Delaunay ball is to create m triangles $\sigma_i\sigma_{(i+1)\%m}p_{n+1}$, $1 \le i \le m$ that all contain the new point p_{n+1}. This procedure indeed produces the desired Delaunay triangulation. DT_{n+1}.

All triangles that are not in $\mathscr{C}(T_n, p_{n+1})$ remain in DT_{n+1}. Those triangles (e.g. $\sigma_i\sigma_{i+1}, p_j$ on Figure 2.4) are Delaunay triangles in DT_{n+1} because their circumcircles neither contain any point of S_n (DT_n is a Delaunay triangulation) nor contain p_{n+1} because they do not belong to $\mathscr{C}(T_n, p_{n+1})$. This implies that edges $\sigma_i\sigma_{i+1}$ are locally Delaunay because the circumcircle of $\sigma_i\sigma_{i+1}, p_j$ do not contain p_{n+1}. The local Delaunaynesss being symetric, it implies that circumcircle of triangle $\sigma_i\sigma_{i+1}, p_{n+1}$ do not contain p_j which proves that every edge of DT_{n+1} is locally Delaunay. Then, DT_{n+1} is the Delaunay triangulation.

2.2 The Bowyer-Watson algorithm

The Bowyer-Watson algorithm is a method for computing the Delaunay triangulation of a finite set of points S in any number of dimensions. It uses the Delaunay kernel in an incremental fashion: starting with an initial triangulation DT_0, points of S are inserted one by one in the triangulation

$$\mathrm{DT}_i = \mathrm{DT}_{i-1} - \mathscr{C}(\mathrm{DT}_{i-1}, p_i) + \mathscr{B}(\mathrm{DT}_{i-1}, p_i), \quad i = 1, \ldots, n.$$

The choice of an initial triangulation DT_0 has to be made.

2.2.1 Super-triangles

The initial Delaunay triangularion DT_0 is composed of 1 or 2 or more "super-triangles". The super-triangles cover the entire convex hull $\Omega(S)$. Super triangles contain points $S_0 = \{p_{-1}, p_{-2}, \ldots, p_{-m}\}$ that do not belong to S (see Figure 2.5).

Points p_j, $1 \leq j \leq n$ are inserted one after the other in the triangulation using the Delaunay kernel (2.1). The final result is a Delaunay triangulation $DT(S \cup S_0)$ of

$$S \cup S_0 = \{p_{-1}, p_{-2}, \ldots, p_{-m}, p_{-1}, p_1, p_2, \ldots, p_n\}.$$

A naive way to recover $DT(S)$ would be to remove from $DT(S \cup S_0)$ every triangle that contains oints of S_0. In reality, the remaining triangles do dot always form the $DT(S)$. On Figure 2.6, triangle $p_k p_j p_l$ should be present in $DT(S)$. Yet, its circumcircle contains point p_{-1} which does not belong to S.

The easiest way of addressing that problem is simply not to fix it. In many situations, $DT(S \cup S_0)$ is a valid input for further use. The is the case for mesh generation.

Yet, one may be interrested in building $DT(S)$. In this case, some modifications to the algorithm have to be made. On Figure 2.6, triangle $p_k p_j p_l$ has its circumcircle that contains p_{-1} and so edge $p_j p_{-1}$ belongs to the Delaunay triangulation. Disappointingly, triangle $p_k p_j p_l$ belongs to $DT(S)$. Triangle $p_k p_j p_l$ would be a Delaunay triangle if p_{-j} was sufficiently far i.e. out of the circumcircle of $p_k p_j p_l$. In this specific case, increasing slightly the size of the super-triangles would do the job but it is not clear how to chose *a priori* the size of the super-triangles that would ensure that any triangle that has an edge on the convex hull has its circumcircle that do not contain any of the p_{-j}'s. Some triangles may be arbitrary flat and their circumcircle arbitrary large. it is indeed impossible to decide *a priory* the right size of the super-triangles.

The easiest solution to recover $DT(S)$ is to start from $DT(S \cup S_0)$ and to apply edge flips in a specific fashion. Assume here that every point p_{-j} is far enough so that it does not fall into any circumcircle. Consider every edge $p_{-i} p_j$ that connectes a point of negative index to a point of positive index. Edge $p_{-i} p_j$ is flippable if it intersects $p_k p_l$. If $p_{-i} p_j$ is flippable, then it should be flipped because triangle $p_k p_j p_l$'s circumcircle does not contain p_{-1}. The principle is to replace an edge of inifinite length with points of finite lenhgth. Note that an edge like $p_i p_{-2}$ should not be flipped because it would create another edge of infinite length. Applying flips successively in that fashion, allows to recover $DT(S)$.

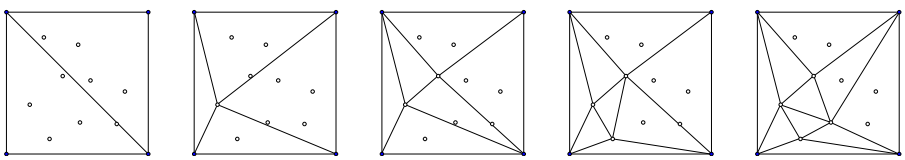

Figure 2.5: A set of 9 points and the two "super-triangles" that contains them all (left). Next Figures show the state of the triangulation after the insertion of 1, 2, 3 and 4 points.

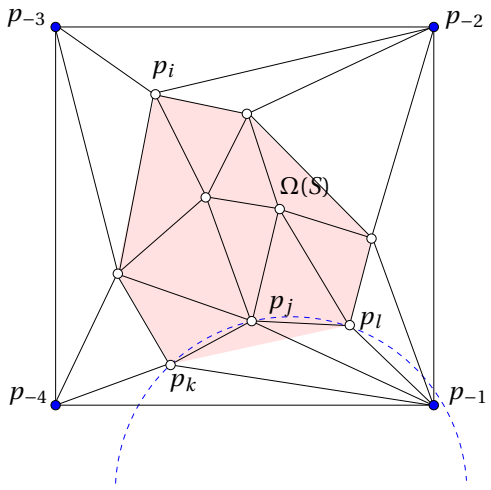

Figure 2.6: Left Figure shows the final triangulation $\mathrm{DT}(S_0 \cup S)$. The convex hull $\Omega(S)$ is shaded and triangles $\mathrm{DT}(S_0 \cup S)$ do not cover it: $\mathrm{DT}(S) \notin \mathrm{DT}(S_0 \cup S)$.

2.2.2 What if $p_{n+1} \notin \Omega(S_n)$?

TODO: explain gift wrapping stuff.

2.3 A robust implementation in $\mathcal{O}(n \log n)$ complexity

Algorithm 1 describes a basic implementation of the Bowyer-Watson algorithm. It has actually two major flaws.

Algorithm 1 is slow: it has a $\mathcal{O}(n^2)$ complexity: at each iteration i, every triangles of DT_{i-1} j is asked if p_i is inside its circumcircle. There is about $2i$ triangles at iteration i which leads to a $\mathcal{O}(n^2)$ complexity. Centers of circumcircles could be computed in advance and stored in the datastructure in order to accelerate the process. Nevertheless, this approach remains quadratic in complexity.

Algorithm 1 suffers from another more subtle flaw that is essentially due to round-off errors. We have assumed that points were in general positions so that no quadruplets of points are cocircular. This hypothesis is indeed not verified in practice: there are numerous applications where circles are involved and where way more than 4 points sit on the same circle. Algorithm 1 could be in trouble because some point may neither be inside nor outside a circulcircle. One solution is to randomly perturbate the position of the points in order to enforce them to be in general position. Here, the question is what is the smallest perturbation that ensures the triangulation process terminates with success.

The first issue can be solved choosing some adequate datastructures and algorithms. The second issue can be addressed by designing essentially two robust pred-

Algorithm 1: Bowyer and Watson's algorithm that creates DT(S)

input : A set of $n+4$ points $S = \{p_{-4}, p_{-3}, p_{-2}, p_{-1}, p_1, \ldots, p_n\} \subset \mathbb{R}^2$
output: The Delaunay triangulation DT(S)

initialize a triangulation data structure DT_0 with 2 super-triangles p_{-1}, p_{-2}, p_{-3} and p_{-2}, p_{-1}, p_{-4}.;

for $i = 1$ **to** n **do**
  for $j = 1$ **to** *size*(DT_{i-1}) **do**
    τ_j is the j^{th} triangle of DT_{i-1};
    if τ_j*'s circumcircle contains* p_i **then**
      Add τ_j to Delaunay cavity $\mathscr{C}(\mathrm{DT}_{i-1}, p_i)$;
      Remove τ_j from DT_{i-1};
    for $j = 1$ **to** *size*($\mathscr{C}(DT_{i-1}, p_i)$) **do**
      τ_j is the j^{th} triangle of $\mathscr{C}(\mathrm{DT}_{i-1}, p_i)$.;
      for $k = 1$ **to** 3 **do**
        e_{jk} is the k^{th} edge of the τ_j. ;
        if e_{jk} *is not shared by any other triangles of* $\mathscr{C}(DT_{i-1}, p_i)$ **then**
          Add a new triangle e_{jk}, p_i into DT_{i-1};

icates.

2.3.1 Robust predicates

Consider three points $a(x_a, y_a)$, $b(x_b, y_b)$ and $c(x_c, y_c)$. The *orientation test*

$$\mathcal{O}_?(a, b, c)$$

determines whether a lies to the left of, to the right of, or on the line L_{bc} defined by points b and c. The orientation test $\mathcal{O}_?$ consist in computing the orientation of triangle abc i.e. to compute:

$$\mathcal{O}_?(a, b, c) = \mathrm{sign}\big((x_a - x_c)(y_b - y_c) - (y_a - y_c)(x_b - x_c)\big).$$

The orientation test is useful in many situations. First, it allows to compute the orientation of a triangle, which is useful by itself. It also allows to verify if two edges ab and cd intersect, which is the case if

$$\mathcal{O}_?(a, b, c) \times \mathcal{O}_?(a, b, d) < 0 \text{ and } \mathcal{O}_?(c, d, a) \times \mathcal{O}_?(c, d, b) < 0.$$

The computation of the orientation test $\mathcal{O}_?(a, b, c)$ looks very simple (7 operations). Some interresting issues appear yet when a is sufficiently close to line bc. As an example [**?**], consider

$$b(12, 12), \ c(24, 24), \text{ and } a(1/2 + i\varepsilon), 1/2 + j\varepsilon),$$

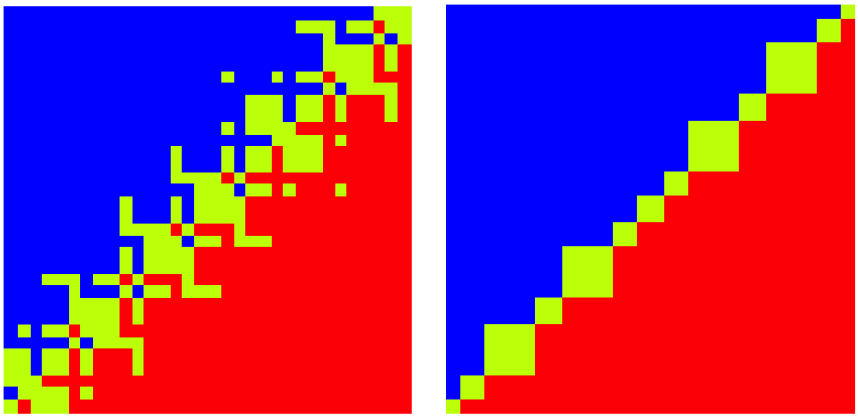

Figure 2.7: Strange behavior of the orientation test. Left figure shows $\mathcal{O}_?(a,b,c)$ for $b(12,12)$, $c(24,24)$, and $a(1/2+i\varepsilon), 1/2+j\varepsilon)$, $\varepsilon = 2^{-53}$ and $0 \leq i,j \leq 2^8$. Right figure shows $\mathcal{O}_?(c,b,a)$. .

$\varepsilon = 2^{-53}$ and $0 \leq i,j \leq 2^8$. Note that 2^{-53} is the significand precision of a double-precision.

In Figure 2.7 the 256^2 results of the $\mathcal{O}_?$ were reported on a $2D$ graph. Green dots are for $\mathcal{O}_?(a,b,c) = -1$, red dots are for $\mathcal{O}_?(a,b,c) = 1$ and yellow dots are for $\mathcal{O}_?(a,b,c) = 0$. We should only see yellow dots only on the diagonal of the square. This is obviously not the case: the orientation test behaves randomly when points are close to be aligned. The result of $\mathcal{O}_?$ is wrong even for points that are at 20 times the significand precision away from the diagonal. Even worse: results obtained with $\mathcal{O}_?(c,b,a)$ should be the same as the ones for $\mathcal{O}_?(a,b,c)$. The second graph proves that this is far from being true. This strange behavior is due to roundoffs. A robust way of computing the orientation test requires more precise (and more expensive) floating-point arithmetics. It is of course too expensive to compute every predicate in an exact fashion. Static filtering consist in assuming that $\mathcal{O}_?$ gives the right answer if

$$|\mathcal{O}_?| > \epsilon \times (\max(x_a, y_a, x_b, y_b, x_c, y_c))^2.$$

In [**?**], authors show that $\epsilon = 10^{-15}$ is considered as secure for the 2D orientation test. This value is verified experimentally on Figure 2.7. If $|\mathcal{O}_?|$ is too small, arbitrary precision arithmetic on floating-point numbers is applied (we use here the GNU Multiple Precision Floating-Point Reliable Library [**?**] that allows to to choose the precision of the computations). Double-precision floating point numbers have a precision of 53 bits (16 significant digits). We have implemented $\mathcal{O}_?$ using 200 bits i.e. with about 60 significand digits! High precision floating point arithmetics coupled with a static filter ($\epsilon = 10^{-15}$) allow to produce the excpected results (see Figure 2.8). The strange behavior of the orientation test of Figure 2.7 has completely disappeared. Only points on the diagonal of the square are considered to be on line bc.

There is another useful predicate for Delaunay triangulations: the *incircle test*.

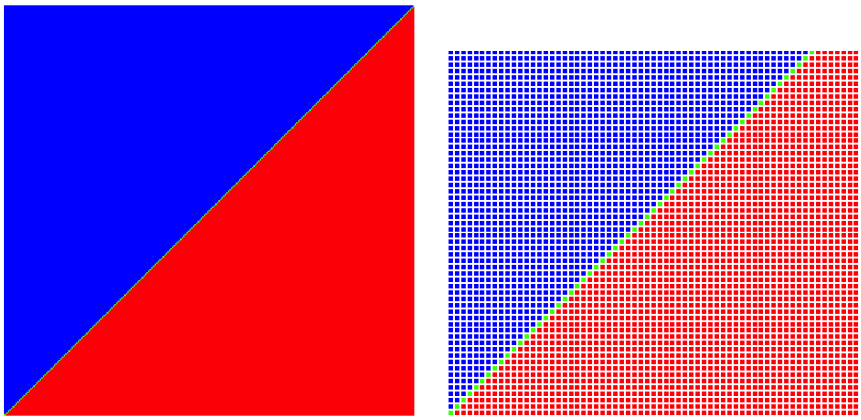

Figure 2.8: $\mathcal{O}_?(a,b,c)$ using a robust predicate. Right Figure is a zoom.

Consider three points $a(x_a, y_a)$, $b(x_b, y_b)$, $c(x_c, y_c)$ and $d(x_d, y_d)$. The incircle test

$$\mathcal{C}_?(a,b,c,d).$$

determines whether d lies inside the circle defined by points a, b, and c. The incircle test has the same robustness issues as the orientation test. Bowyer-Watson algorithm 1 that uses a non robust circle test can possibly produce invalid meshes.

A strategy that couples static filtering ($\epsilon = 10^{-11}$) and high precision floating point arithmetics result in a robust incircle test.

2.3.2 More on Adjacencies

A triangulation is composed of a collection of "entities" (triangles, edges and points) together with their adjacencies. Any entity bounds and/or is bounded by other ones of higher and/or lower dimension. This *adjacency information* represents the graph of a mesh. All these adjacency sets do not need to be present in a given representation. Moreover, some entities may simply not be present: the explcit representation of the edges of a triangulation is not relevant in many situations (the Delaunay kernel e.g.).

It is interesting at this point to gather some statistics about the average number of adjacencies per entity that occurs in triangulations. With these statistics, we will be able to compute the cost of a given representation i.e. its size in the memory of a computer.

Consider a triangulation $T(S)$ with $S = \{p_1, \ldots, p_n\}$ that have n_h vertices on its convex hull $\Omega(S)$. If n is the number of vertices in T, then we already know that the number of triangles of T is $n_f = 2(n-1) - n_h$ and the number of edges of T is $n_e = 3(n-1) - n_h$. (see proposition 1.2.1). In many cases, algorithms that deal with triangulation have to keep track of "upward adjacencies" i.e. the set of triangles or of edges that are adjacent to a given vertex or the triangles that are adjacent to an

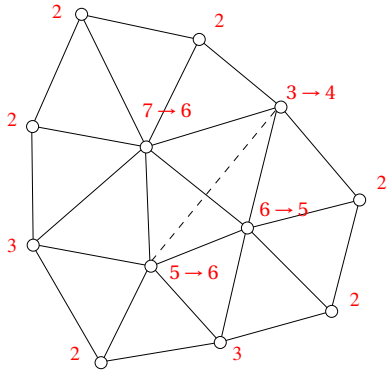

Figure 2.9: A triangulation T with $n = 12$ and $n_h = 9$. The average number of triangles adjacent to a vertex is (see (2.2)) $n_{vf} = 6 - \frac{3\times 9+6}{12} = 3,25$. This average can also be computed explicitely: $n_{vf} = \frac{39}{12} = 3,25$.

edge. The number of triangles and edges adjacent to a vertex are called respectively n_{vf} and n_{ve}.

Proposition 2.3.1 *Consider a triangulation T with n points and n_h points on its convex hull $\Omega(T)$. We claim here that a point of T has in average $n_{vf} = 6 - \frac{3n_h+6}{n}$ adjacent triangles and $n_{ve} = 6 - \frac{2n_h+6}{n}$ adjacent edges.*

Proof A triangle is adjacent to three vertices and a vertex is adjacent to n_{vf} triangles. This leads to

$$n_{vf} n = 3n_f = 3\,(2(n-1) - n_h)$$

and we have the result

$$n_{vf} = 6 - \frac{3n_h + 6}{n}. \tag{2.2}$$

Similarly, n_{ve} that is the number of edges adjacent to a vertex can be computed as

$$n_{ve} n = 2n_e = 2\,(3(n-1) - n_h)$$

which gives

$$n_{ve} = 6 - \frac{2n_h + 6}{n}. \tag{2.3}$$

∎

2.3.3 Choice of a datastructure

Figure 2.9 illustrate equation (2.2). The average number of adjacencies per entity in the triangulation is know in advance. Yet, as it is seen on Figure (2.9), this number

varies from one vertex to another. This number may also change locally: an edge flip removes one triangle of the adjacency of the two vertices of the edge that is flipped and adds one triangle tho the adjacency of the two vertices of the new edge (see Figure (2.9)). The number of upward adjacencies of a given vertex may change which implies that datastructures that would keep track of such adjacencies should be of variable size.

When the size of data may vary, memory allocation has to be used, which implies indirect memory access and extra data storage. Datastructures of fixed size are always preferred. Yet, some kind of upward adjacencies should exist in the datastructure in order to accesss neighborhood of a mesh entity withoud traversing the whole triangulation.

There exist one type of upward adjacency that is of fixed size: there is either 1 or 2 adjacent triangle to an edge. Note that this hypothesis implies that the triangulation is *manifold* i.e. each edge is shared by no more than 2 faces. This is a common assumption for many algorithms and we will assume the triangulation to be manifold for now.

At this point, we'd like to choose how we will represent our triangulation on a computer. The problem of choosing a datastructure is crucial. A good datastructure is has a low memory footprint but allows to compute local adjacencies in constant time.

Technically, an adjacency is implemented as a pointer (the address of the adjacent entity). Moreover, if a given entity (point, edge or face) is explicitely represented in a datastructure, it also requires one pointer (the address of the entity).

It is interresting to cout the total amount of pointers N_p that are required in a given mesh representation (i.e. for a given datastructure). In what follows, we assume that $n \gg n_h$ and $n \gg 1$ which implies that $n_e \simeq 3n$, $n_f \simeq 2n$, $n_{ve} \simeq 6$, $n_{vf} \simeq 6$ and $n_{ef} \simeq 2$.

A naive choice could be to store all entities and all their adjacencies. This datastructure is said to be full for obvious reasons. The number of pointers that is required is

$$N_p = n(1 + n_{ve} + n_{vf}) + n_e(2 + 1 + n_{ef}) + n_f(3 + 3 + 1) \simeq 42n.$$

The full datastructure is clearly overkill in term of memory. Moreover, using such a datatrsucture in algorithms requires complicated updates which makes that approach totally ininterresting.

Another choice is the bidirectional datastructure [**?**]. In this datastructure, vertices keep track of their adjacent edges, edges know about their adjacent vertices and faces and face know about their edges. This datastructure is complete in the sense that all entities are represented explicitely and that any adjacency information can be recovered using local searches. The number of pointers that is required is

$$N_p = n(1 + n_{ve}) + n_e(2 + 1 + n_{ef}) + n_f(3 + 1) \simeq 30n.$$

This is again a very heavy datastructure that requires complex updates while used in algorithms.

In many cases, the only information that is required in a representation is the list of vertices of a triangle. This is the case in most of the finite element formulations or

```
struct Vertex {
  double x,y,z;
  Vertex (double X, double Y, double Z) :
    x(X), y(Y), z(Z) {}
};
```

Listing 2.1: Vertex Datastructure

to draw the mesh. Here, the number of pointers that is required is

$$N_p = n + n_f(3+1) \simeq 9n.$$

This is clearly the minimum amount of information possible. No upward adjacenci is available here so that it is impossible to devise efficient meshing algorithms with such a datastructure.

Most popular data structures for storing adjacency information of polygonal meshes are edge-based. Winged-edge [**?**] and half-edge [**?**] datastructures apply to manifold meshes while Winged-edge and half-edge data structure uses edges to keep track almost everything. In a winged-edge datastructure, each edge stores 8 pointers to neighboring edges, faces and points. Faces and points store one pointer, so that the number of pointers that is required is

$$N_p = n(1+1) + n_e(1+8) + n_f(1+1) = 33n.$$

The advantage of such a datatrsucture is that it is easy to update when local operations are performed. Yet, it is quite heavy.

Those datastructures are suboptimal for algorithms like the Delaunay triangulation. Representing edges explicitely is not mandatory here and edges are the entities that are the most numerous in a triangulation. In this text, we use a datastructure that is face-based: each triangle knowns about its 3 vertices and its 3 neighboring triangles. Each vertex knows about its coordinates. That's pretty much all. The number of pointers that is required is

$$N_p = n + n_f(1+6) = 15n.$$

This datastructure is way lighter than edge-based ones. With 8 bytes pointers, the memory footprint of a mesh with $n = 10^6$ is 120 Mb. Another advantage is that it can be extended in 3D, which is not the case for edge-based datastructures.

Vertex datastructure

The vertex datastructure is quite simple: a vertex knows about its coordinates (see Listings 2.1.

Edge datastructure

Even though we do not maintain edges of the triangulation in our algorithms, it is sometimes necessary to build edges for a subset of triangles of the triangulation.

```
struct Edge {
   Vertex *vmin,*vmax;
   Edge (Vertex *v1, Vertex *v2)
      vmin = std::min (v1,v2);
      vmax = std::max (v1,v2);
   }
   bool operator < (const Edge &other) const {
      if (vmin < other.vmin) return true;
      if (vmin > other.vmin) return false;
      if (vmax < other.vmax) return true;
      return false;
   }
};
```

Listing 2.2: Edge Datastructure

The edges that we consider are nort oriented: they are equal if they connect the same two vertices. The datastructure shown in Listings 2.2 allows to construct an edge with two vertices and to compare two edges (edges are compared comaring their vertex pointers in a lexicographic manner).

Face datastructure

The triangles are maintained in the triangulation. Each triangle maintains its three vertices and its three neighbors. We assume that neighbor `F[k]` is the triangle that is on the other side of edge with vertices `V[k]` and `V[(k+1)%3]`. We also assume that we have a function `inCircle` that predicts if vertex `V` is inside the circumcircle of the `Face` and a function `centroid` that computes the centroid of the face. The Face datastructure is shown in Listings 2.3.

2.3.4 Algorithms

A local mesh modification works as follows. A cavity of triangles is removed from the mesh (see Figure 2.10). The cavity is remeshed and mesh datastructures are updated in order to take into account the modification. More specifically, each new `Face` of the remeshed cavity has to be connected to its neighboring faces. Those neighboring faces may be new as well or may be neighboring triangles of the cavity.

Algorithm depicted in Listings 2.4 is the building block of all other algorithms that are performing local mesh modifications: `computeAdjacencies` computes adjacencies of a list of N triangles. It has a $\mathcal{O}(N \log N)$ complexity (one search/insert on a `std::map` per triangle). We use here some associative containers from the standard template library.

Another important building block in our implementation is the computation of the Delaunay cavity. We assume here that one initial triangle t has been found that has its circumcircle containing a given vertex. Algorithm in Listings 2.5 allows to

```
struct Face {
    Face    *F[3];
    Vertex  *V[3];
    bool deleted;
    Face (Vertex *v0, Vertex *v1, Vertex *v2){
       V[0] = v0; V[1] = v1; V[2] = v2;
       F[0] = F[1] = F[2] = NULL;
       deleted = false;
    }
    Edge getEdge (int k) {
       return Edge (V[k],V[(k+1)%3]);
    }
    bool inCircle (Vertex *c);
    Vertex centroid ();
};
```

Listing 2.3: Face Datastructure

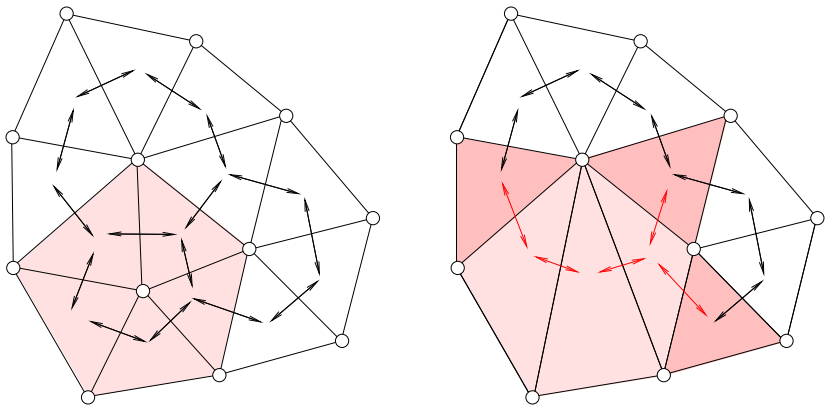

Figure 2.10: A cavity (left figure in light pink) is removed from the meh. It is remeshed (left figure in light pink). Adjacencies (double arrows) are updated (red double arrows) for all new triangles (light pink) as well as for all neighboring triangles of the cavity (dark pink).

```
void computeAdjacencies (std::vector<Face*> &cavity) {
  std::map < Edge , std::pair < int , Face* > >edgeToFace;
  for (int iFace=0 ; iFace < cavity.size() ; iFace++ ){
     for (int iEdge=0 ; iEdge < 3 ; iEdge++){
        Edge edge = cavity[iFace]->getEdge(iEdge);
        std::map < Edge , std::pair < int , Face* > >::iterator it =
            edgeToFace.find(edge);
        if (it == edgeToFace.end()){
           // edge has not yet been touched, so create an entry
           edgeToFace.insert(std::make_pair (edge,
              std::make_pair(iEdge, cavity[iFace])));
         }
         else{
          // Connect the two neighboring triangles
           cavity[iFace]->F[iEdge] = it->second.second;
           it->second.second->F[it->second.first] =  cavity[iFace];
          // Erase edge from the map
           edgeToFace.erase(it);
         }
      }
   }
}
```

Listing 2.4: An algorithm for connecting triangles in a cavity

compute the Delaunay cavity using a depth-first search technique. The theory ensures that the Delaunay cavity is simply connected: triangles that form the Delaunay cavity are neighbors of t, neighbors of the neighbors of t and so on. The neighborhood of t is searched recursively until a triangle is found that is valid i.e. that does not violate the empty circumcircle property. Triangles that have been checked are marked as deleted to avoid infinite loops. Two other outputs are computed that will serve us in constructing the Delaunay ball and in computing adjacencies. The set of edges that form the boundary of the cavity is also computed. The corresponding valid triangles that are on the other side of the boundary of the Delaunay cavity are also computed.

Computing the Delaunay cavity requires a seed triangle i.e. a triangle t of the triangulation that is invalid. The last bit algorithm that is provided here allows to perform a search in a mesh along a given direction and find the desired triangle. Triangulations we are dealing with cover the convex hull $\Omega(S)$ of the set of points S. So, if c is the centroid of a given triangle t and if $p \in S$ is a target point, line cp is entirely inside the triangulation and it is possible to find a path of triangles that connect t to the triangle t' that contains p. Algorithm in Listings 2.6 strats from a given triangle and traverses the mesh until an invalid triangle is found. It assumes that a robust orientation test $\mathcal{O}_?$ function is available. Assume a triangulation with n_f triangles, the complexity of algorithm `lineSearch` is at most linear. Asymptotically, it is not absurd to guess that only $\mathcal{O}(\sqrt{n_h})$ triangles will be touched by `lineSearch`

```
void delaunayCavity (Face *f, Vertex *v, std::vector<Face*> &cavity,
                     std::vector<Edge> &bnd, std::vector<Face*>
                        &otherSide){
  if (f->deleted)return;
  f->deleted = true; // Mark the triangle
  cavity.push_back(f);
  for (int iNeigh=0; iNeigh<3 ; iNeigh++){
    if (f->F[iNeigh] == NULL){
      bnd.push_back(f->getEdge(iNeigh));
    }
    else if (!f->F[iNeigh]->inCircle(v)){
      bnd.push_back(f->getEdge(iNeigh));
      if (!f->F[iNeigh]->deleted){
          otherSide.push_back(f->F[iNeigh]);
          f->F[iNeigh]->deleted = true;
      }
    }
    else delaunayCavity (f->F[iNeigh], v, cavity,bnd,otherSide);
  }
}
```

Listing 2.5: An algorithm for computing the Delaunay cavity

```
Face* lineSearch (Face *f, Vertex *v){
    while(1){
      if (f == NULL) return NULL; // we should NEVER return here
      if (f->inCircle(v)) return f;
      Vertex c = f->centroid();
      for (int iNeigh=0; iNeigh<3 ; iNeigh++){
        Edge e = f->getEdge (iNeigh);
        if (orientationTest (&c,v,e.vmin) *
            orientationTest (&c,v,e.vmax) < 0 &&
            orientationTest (e.vmin, e.vmax, &c) *
            orientationTest (e.vmin, e.vmax, v) < 0) {
          f = f->F[iNeigh];
          break;
       }
    }
 }
```

Listing 2.6: An algorithm that finds a invalid triangle

```
void delaunayTrgl (std::vector<Vertex*> &S, std::vector<Face*> &T){
  for (int iP=0 ; iP < S.size() ; iP++ ){
    Face * f = lineSearch ( T[0] , S[iP]);
    std::vector<Face*> cavity;
    std::vector<Edge> bnd;
    std::vector<Face*> otherSide;
    delaunayCavity (f, S[iP], cavity, bnd, otherSide);
    if(bnd.size() != cavity.size() + 2) throw;
    for (int i=0; i<cavity.size(); i++) {
      // reuse memory slots of invalid elements
      cavity[i]->deleted = false;
      cavity[i]->F[0] = cavity[i]->F[1] = cavity[i]->F[2]  = NULL;
      cavity[i]->V[0] = bnd[i].V[0];
      cavity[i]->V[1] = bnd[i].V[1];
      cavity[i]->V[2] = S[iP];
    }
    unsigned int cSize = cavity.size();
    for (int i=cSize; i<cSize+2; i++) {
      Face *newf = new Face (bnd[i].V[0],bnd[i].V[1],S[iP]);
      T.push_back[newf];
      cavity.push_back(newf);
    }
    for (int i=0;i<otherSide.size();i++)
      if (otherSide[i]) cavity.push_back(otherSide[i]);
    computeAdjacencies (cavity);
  }
}
```

Listing 2.7: An algorithm for computing the Delaunay triangulation

which reduce its complexity in practice.

Algorithm in Listings 2.7 is a C++ version of 1. It has clearly a worst complexity of $\mathcal{O}(n^2)$ but could possibly behave better i.e. like $\mathcal{O}(n^{3/2})$. It starts with an initial triangulation made of some super triangles that cover the convex hull of S and inserts the points incrementally.

The code that is provided here is actually working as is. We have used it to compue Delaunay triangulations of random points. The following table presents results of the algorithm for a set of n random points in the plane that have been inserted in a random order.

In Table 2.1, N_{search} is the average number of serarches that have been performed in `lineSearch` and N_{cavity} is the average size of the Delaunay cavity. Even though this implementation may not be optimal, it shows the basic features of the algorithm. First, the average cavity size is asymptotically optimal: a cavity of size 4 produce 6 new triangles adjacent to a vertex which is what the theory predicts. Then the number of walks that the `lineSearch` algorithm increases approximatively like the square root of n: $56.8 \times \sqrt{10} = 179.6$ which is close to 161. The complexity of the

n	10^3	10^4	10^5	10^6
N_{search}	19.8	56.8	161	503
N_{cavity}	3.85	3.97	3.99	3.99
t(sec)	0.012	0.198	4.85	172

Table 2.1: Results of the `delaunayTrgl` algorithm applied to random points.

algorithm written as is is close to $\mathcal{O}(n^{3/2})$. For large meshes, the $\mathcal{O}_?$ predicate takes about 50% of the CPU time so that the most significant part if the time is spend in searching for an initial triangle.

The bottleneck of the Delaunay triangulation as it is written in `delaunayTrgl` is the increasing effort that has to be done at each point insertion to find a triangle seed for building the Delaunay cavity.

Assume now that we are able to sort the set of points S in such a way that two successive points in the list would be close to each other. In Algorithm `delaunayTrgl`, we take as initial guess the first triangle of the list and search into the domain. We could chage that by choosing one of the triangles of the cavity that is associated to the vertex that was inserted previously in the list.

2.3.5 Hilbert Curves

A curve $x(t)$ is defined as the mapping

$$x(t), [0,1] \rightarrow x \in \mathbb{R}^3.$$

Curves are perceived as one dimensional objects. Yet, it can be shown that a continuous curve can pass through every point of a unit square. The Hilbert space filling $\mathscr{H}(t)$ curve is a one dimensional curve which visits every point within a two dimensional space. It may be thought of as the limit

$$\mathscr{H}(t) = \lim_{k\rightarrow\infty} \mathscr{H}_k(t)$$

of a sequence of curves $\mathscr{H}_k$. Curves $\mathscr{H}_1$ and $\mathscr{H}_2$ are depicted on Figure 2.11. There are lots of references that show how to actually draw Hilbert curves: this is a distraction from the essential property of the curve, and its importance to mesh generation.

Hilbert curves provide an ordering for points on a plane. Forget about how to connect adjacent sub-curves, and instead focus on how we can recursively enumerate the quadrants.

A local frame is associated to each quadrant: it consist in its center x_0 two orthogonal vectors b and r (see Figure 2.11). At the root level, enumerating the points is simple: proceed around the four quadrants, numbering them

$$(0) = x_0 - \frac{b+r}{2} \quad (1) = x_0 + \frac{b-r}{2} \quad (2) = x_0 + \frac{b+r}{2} \quad (3) = x_0 - \frac{b-r}{2}.$$

We want to determine the order we visit the sub-quadrants while maintaining the overall adjacency property. Examination reveals that each of the sub-quadrants

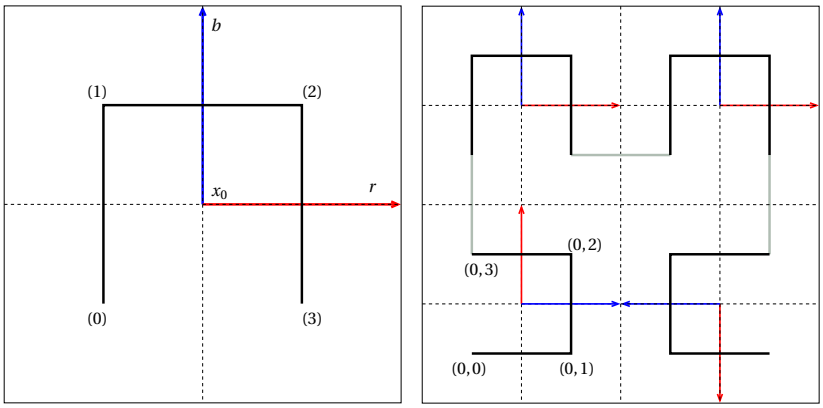

Figure 2.11: Curves $\mathscr{H}_1$ and $\mathscr{H}_2$.

curves is a simple transformation of the original pattern. Figure 2.11 illustrate the first level of that recursion.

Quadrant (0) is itself divided into four quadrants $(0,0)$, $(0,1)$, $(0,2)$ and $(0,3)$. Its center is simply set to (0) and two vectors b and r are changed as

$$b \leftarrow r/2 \text{ and } r \leftarrow b/2.$$

For quadrant $(0,1)$ and $(0,2)$ we have

$$b \leftarrow b/2 \text{ and } r \leftarrow r/2.$$

and finally for quadrant $(0,3)$:

$$b \leftarrow -r/2 \text{ and } r \leftarrow -b/2.$$

creates 4 sub quadrants. If we consider a maximal recursion depth of d, each of the final subquadrants will be assigned to a set of d "coordinates" i.e. $(k_0, k_1, \ldots, k_d)$, k_j being 0,1,2 or 3.

Algorithm in Listings 2.8 compute the Hilbert coordinates of a given point x, y, starting from an initial quadrant define by its center x_0, y_0 and two orthogonal directions. Each point x of $\mathbb{R}^2$ has its coordinates on the Hilbert curve. Sorting a point set with respect to Hilbert coordinates allow to ensure that two successive points of the set are close to each other. In the context of the Bowyer-Watson algorithm, this kind of data locality could potentially decrease the number of local searches N_{search} that were required to find the next invalid triangle.

Algorithm 2.8 was used to sort sets of 1000 and 10000 points. The results are presented on Figure 2.12. On the Figure, two successive points in the sorted list are linked with a line.

The main cost of sorting points is on the sorting algorithm itself and not on the computation of the Hilbert curve coordinates: sorting over a million points takes

```
void HilbertCoord( double x,    double y,    double x0,    double y0,
                   double xRed, double yRed, double xBlue, double yBlue,
                   int d,       int bits[] ){

  for (int i = 0; i <d; i++) {
    double coordRed  = (x-x0) * xRed  + (y-y0) * yRed;
    double coordBlue = (x-x0) * xBlue + (y-y0) * yBlue;
    xRed/=2; yRed/=2;       xBlue/=2; yBlue/=2;
    if (coordRed <= 0 && coordBlue <= 0) { // quadrant 0
      x0 -= (xBlue+xRed);       y0 -= (yBlue+yRed);
      swap (xRed,xBlue);        swap (yRed,yBlue);
      bits[i] = 0;
   }
    else if (coordRed <= 0 && coordBlue >= 0) { // quadrant 1
      x0 += (xBlue-xRed);       y0 += (yBlue-yRed);
      bits[i] = 1;
    }
    else if (coordRed >= 0 && coordBlue >= 0) { // quadrant 2
     x0 += (xBlue+xRed);       y0 += (yBlue+yRed);
     bits[i] = 2;
    }
    else if (coordRed >= 0 && coordBlue <= 0) { // quadrant 3
      x0 += (-xBlue+xRed);     y0 += (-yBlue+yRed);
      swap (xRed,xBlue);       swap (yRed,yBlue);
      xBlue = -xBlue;          yBlue = -yBlue;
      xRed = -xRed;            yRed = -yRed;
      bits[i] = 3;
    }
  }
}
```

Listing 2.8: An algorithm for computing Hilbert coordinates

n	10^3	10^4	10^5	10^6
N_{search}	2.34	2.46	2.50	2.50
N_{cavity}	4.06	4.13	4.16	4.17
t(sec)	0.0097	0.090	0.92	9.2

Table 2.2: Results of the `delaunayTrgl` algorithm applied to random points. Points were initially sorted through using a Hilbert sort.

less than a second on a standard laptop. Table 2.2 present timings and statistics for the same point sets as in table 2.1, but while having sorted the points S using the Hilbert curve. The number of serarches is not increasing anymore with the size of the set. This is important: the complexity of the Delaunay triangulation algorithm now is linear in time. Of course, sorting points has a $n \log n$ complexity so that the overall process is in $n \log n$ as well. Yet, the relative cost of sorting the points is negligible with respect to the cost of the triangulation itself.

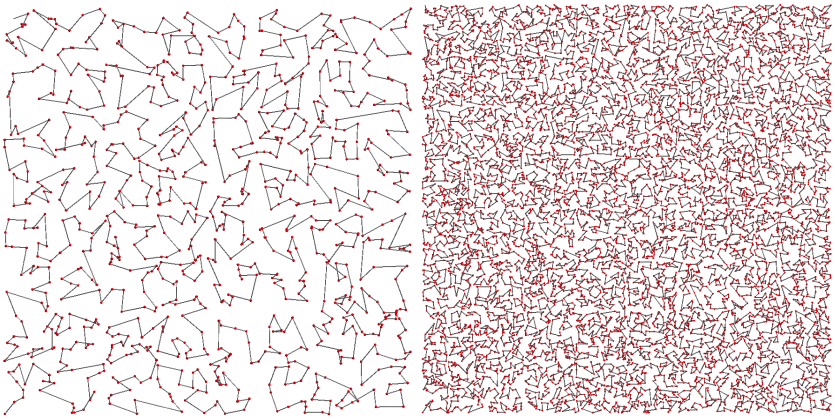

Figure 2.12: Hilbert sort of sets of 1000 and 10000 random points.

Explain brio : The trick is to organize the point set in random buckets of increasing sizes, Hilbert sort being used only inside a bucket. I observe that this is useless in my implementation that do not really care a lot of memory allocation optimization strategies.

2.3.6 Edge flip

TODO: rite the edge flip algorithm and write the algorithm that recovers the Delaunay triangulation DT(S) starting from DT($S_0 \cup S$).